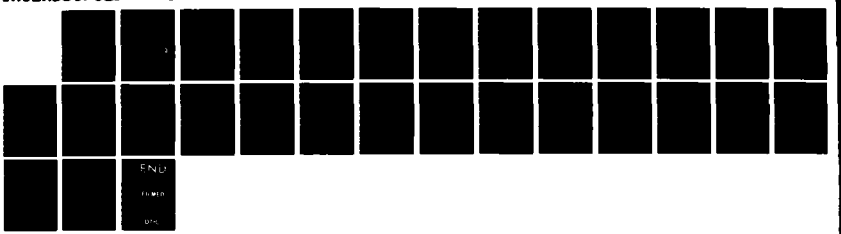
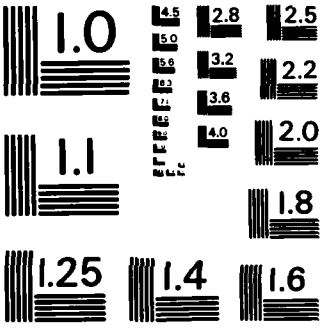
AD-A162 141 FINDING TEST-AND-TREATMENT PROCEDURES USING PARALLEL COMPUTATION(U) DUKE UNIV DURHAM NC DEPT OF COMPUTER SCIENCE L D DUVAL ET AL. OCT 85 CS-1985-23 1/1

UNCLASSIFIED AFOSR-TR-85-1049 AFOSR-83-0205 F/G 12/1 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A

(3)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

(2)

1. REPORT NUMBER	2. GOVT ACCESSION
AFOSR-TR- 85-1049	

AD-A162 141

4. TITLE (and Subtitle)

Finding test-and-treatment procedures using parallel computation

Technical paper

6. PERFORMING ORG. REPORT NUMBER

CS- 1985- 23

7. AUTHOR(s)

L. D. Duval, R. A. Wagner, Y. Han, D. W. Loveland

8. CONTRACT OR GRANT NUMBER(s)

AFOSR 83-0205

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Computer Science Department
Duke University
Durham, North Carolina 27706

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

61102F 2304/A7

11. CONTROLLING OFFICE NAME AND ADDRESS

same as #14

12. REPORT DATE

October 1985

13. NUMBER OF PAGES

23

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

Air Force Office of Scientific Research
Air Force System Command
Bolling AFB
Washington, DC 20332

15. SECURITY CLASS. (of this report)

UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release,
distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

DTIC ELECTE DEC 09 1985 S D E

18. SUPPLEMENTARY NOTES

Submitted for publication

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Parallel algorithms, NP-hard problems, complexity

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A parallel algorithm for the NP-hard problem test-and-treatment is presented for a machine whose number of connections is $\frac{3p}{2}$, where p is the number of processing elements (PEs), and where the PEs are simple enough such that a machine with 2^{20} PEs is currently implementable and 2^{30} PE machine is

DTIC FILE COPY

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract continued

feasible. The speedup of $O(\frac{p}{\log p})$ is realizable because we are able to transform the dynamic programming solution into the ASCEND/DESCEND scheme with considerable attention to the communication problem. This algorithm is realized on the Boolean Vector Machine, a fully designed cube-connected-cycle system where processor allocation and other control issues have been faced. The particular NP-hard problem is of independent interest; it generalizes the binary testing problem by introducing treatments on an equal basis with tests. Applications of this test-and-treatment problem are found in medical diagnosis, systematic biology, machine fault location, laboratory analysis and many other fields.

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

CS-1985-23

Finding Test-and-Treatment Procedures Using Parallel Computation

Louis D. Duval
Robert A. Wagner
Yijie Han
Donald W. Loveland

Department of Computer Science
Duke University

85 12 6 033

Finding Test-and-Treatment Procedures Using Parallel Computation

Louis D. Duval
Robert A. Wagner
Yijie Han
Donald W. Loveland

Department of Computer Science
Duke University
Durham, NC 27706

ABSTRACT

A parallel algorithm for the NP-hard problem test-and-treatment is presented for a machine whose number of connections is $\frac{p^2}{2}$, where p is the number of processing elements(PEs), and where the PEs are simple enough such that a machine with 2^{20} PEs is currently implementable and 2^{30} PE machine is feasible. The speedup of $O(\frac{p}{\log p})$ is realizable because we are able to transform the dynamic programming solution into the ASCEND/DESCEND scheme with considerable attention to the communication problem. This algorithm is realized on the Boolean Vector Machine, a fully designed cube-connected-cycle system where processor allocation and other control issues have been faced. The particular NP-hard problem is of independent interest; it generalizes the binary testing problem by introducing treatments on an equal basis with tests. Applications of this test-and-treatment problem are found in medical diagnosis, systematic biology, machine fault location, laboratory analysis and many other fields.

Index terms – Parallel algorithm, NP-hard problem, complexity.

October 3, 1985

• Work reported herein is partially supported by the Air Force under grant number AFOSR 81-0221 and AFOSR 83-0205.

Finding Test-and-Treatment Procedures Using Parallel Computation

Louis D. Duval

Robert A. Wagner

Yijie Han

Donald W. Loveland

Department of Computer Science

Duke University

Durham, NC 27706

1. Introduction

The test-and-treatment problem originally defined by D.W. Loveland is a generalization of the binary testing problem studied by many researchers(see [1][2][6][7][11]). This problem is of independent interest since it finds applications in medical diagnosis, systematic biology, machine fault location, laboratory analysis and many other fields. A parallel algorithm for this problem is presented which is implemented on the Boolean Vector Machine(BVM), a machine formed by connecting many tiny PEs into a cube-connected-cycle network. The PEs are so small that a machine with 2^{20} PEs is implementable using current VLSI technology, and even 2^{30} PE machine is feasible. By handling the communication problem carefully we are able to transform the dynamic program solution into the ASCEND/DESCEND scheme. This solution to the communication problem and a careful algorithm design for generating control bits solves the PE allocation problem. As a result we are able to achieve $O(\frac{p}{\log p})$ speedup on such a machine with only $\frac{3p}{2}$ connections

* Work reported herein is partially supported by the Air Force under grant number AFOSR 81-0221 and AFOSR 83-0205.

among p PEs.

It has been shown[3][6] that finding optimal solution to the binary testing problem is in general NP-hard; that is, no polynomial algorithm on sequential machines is known. Since the test-and-treatment problem generalizes the binary testing problem, the test-and-treatment problem is also NP-hard. With the development of VLSI technology parallel machines with thousands and even millions of processing elements(PEs) will be available. It is now practically possible to speed up the computation considerably by trading huge number of PEs for speed. Solutions to the NP-complete problems on parallel machines have appeared in the literature[4][8]. Parallel algorithms are considered to be good if the speedup $S = \frac{T_1}{T_p}$ achieved is equal or close to p, where T_i is the time complexity of a parallel machine with i PEs. It is especially of practical interest when these algorithms can be implemented on "practical" parallel machines efficiently, since PE allocation problem must be taken into account and communication between PEs must be handled carefully. In this paper we present a parallel algorithm approach to the problem. The Boolean Vector Machine(BVM) [16] is the parallel computation model we have chosen, a model that uses the Cube-Connected-Cycle(CCC) [13] structure. For the BVM, each processing element(PE) of is connected to three other PEs by a one-bit wide connection path. It has been estimated that a BVM with 2^{20} PEs is feasible. Such a huge parallel machine could be used to solve moderate-sized NP-complete or NP-hard problems. The time complexity and processor complexity of the TT algorithm on this machine model are respectively $O(kp(k+\log N))^*$ and $O(N2^k)$, where k is the size of the universe which is a set of objects containing the malfunctioned one, p is the precision required, N is the total number of the tests and treatments available. This result represents a speedup of $O(\frac{p}{\log p})$, with regard to the known sequential algorithm which could be obtained by modifying the backward induction algorithm given by Garey[1]. The $\log p$ in the speedup is accounted for the communications needed among the PEs. As can be shown by a simple fan-in argument, $\Omega(k+\log N)$ time is required for the communication among $O(N2^k)$ PEs. Considering

* Logarithms in this paper are to the base 2

that each PE in our machine has only three links to other PEs, this log$p$ factor is quite reasonable.

Of particular interest here is the fact that $O(\frac{p}{\log p})$ speedup is shown for a machine so simple in structure that the number of PEs on the order of $2^{30}(\approx 10^9)$ is feasible. For 2^{30} PEs, approximately 15 elements(say, disease candidates) could be processed in parallel (assuming worst case possibilities) to find the best treatment for the true disease even if all possible tests and treatments were available(i.e. $N = O(2^k)$. A speedup of roughly 10^6 could thus be realized over a sequential processing of a test-and-treatment problem with 15 candidates. (This allows for the parallelism of 64 bits that a sequential machine might possess.)

Our algorithm was designed to optimize performance for relatively few tests and treatments, e.g. $N = O(k^b)$, for fixed b. Other approaches are reasonable if $N = O(2^k)$ is commonly used. We note that a few more elements, e.g. 20, can be processed in parallel if $N = O(k^2)$, say.

The test and treatment problem requests the selection of a minimum test and treatment procedure under an expected cost criteria. The problem arises whenever a fault (disease, system malfunction) must be treated. The classic example is medical diagnosis and treatment, but other applications also are important, such as computer system fault location and correction and logistical system breakdown correction. In general, the problem exists whenever a sizable population of complex objects (people, ships, computers) must be maintained at reasonable cost.

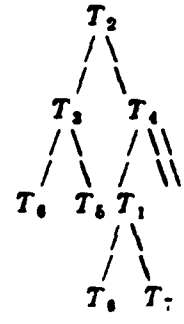
The problem specification consists of a universe $U = \{0,1,...,k-1\}$ of k objects, each with an associated weight P_i, and a set of

$$T_i, 1 \leq i \leq N,$$

tests and treatments, each with an associated cost. The T_i, $1 \leq i \leq m$, denote tests, and the T_i, $m < i \leq N$, denote treatments. We assume that only one object is actually faulty, its identity is unknown, and each object i has *a priori* likelihood P_i of being the faulty object. Each test and treatment is specified by a subset of the universe; if the unknown object is in the test or treatment set then the test responds positively, or "is successful", or the treatment is successful. If the test is successful, one eliminates the other objects from consideration (and if negative, one

eliminates the test set of objects), while a successful treatment ends the procedure. A failed treatment means the processing must continue. A successful TT procedure must provide for each object to be treated; a TT problem specification is *adequate* if there exists a successful TT procedure. With each test and treatment T_i a cost t_i of executing that test or treatment is given with the problem specification.

From the above description we see that a TT procedure is a binary decision tree, with both test and treatment nodes. A typical TT procedure is given in Figure 1, where the single arc is used for both test outcomes (the positive outcome to the left by convention) and a treatment failure, and the double-line arc denotes a treatment set. (The double arc is for emphasis only since every branch of a successful TT procedure must terminate in a treatment set.)



U={0,1,2,3,4,5}, P_i =1/6 for all i,
T_1=2, T_2=0,4, T_3=0,1, T_4=0,1,3,
T_5=4, T_6=0,2. T_7=5, m =3
t_i =1 for all i.

Fig 1.

The TT procedure Tree has an expected cost defined as follows:

$$Cost(Tree) = \sum_{i \in U} (\text{cost of all tests and treatments encountered if i is the faulty object}) \cdot P_i$$

The desired solution is the procedure which minimizes this cost. Thus

$$Cost = \min_{\text{all trees}} Cost(Tree).$$

Rather than enumerate all the possible TT trees and take the minimum cost directly, we use the approach of dynamic programming and note that the optimal tree must apply the minimum cost action (test or treatment) to already optimal subtrees. The optimal subtrees are obtained by beginning with one-object trees and combining trees as just described. For each one-object set S of objects we compute the cost C(S) of the minimum TT tree as follows:

$$C(S) = \min_{m < i \le n} (t_i \cdot p(S))$$

where $p(S) = P_j$ for $S = \{j\}$. There is only a treatment component for one-object sets since the set cannot be split. (Note that we have not normalized the set of weights, so technically these are not TT problems themselves). For an arbitrary non-singleton set S of objects we compute the cost C(S) as follows:

$$C(S) = \min\Big[\min_{1 \le i \le m} (t_i \cdot p(S) + C(S \cap T_i) + C(S - T_i)),$$
$$\min_{m < i \le n} (t_i \cdot p(S) + C(S - T_i))\Big].$$

where $p(S) = \sum_{j \in S} p_j$.

This definition is from first principles: the value t_i is charged to each object subject to that action and the total weight of those objects to be charged is p(S). For tests, one adds in the cost $C(S \cap T_i)$ of the set $S \cap T_i$ to which the test responds positively (the test set) plus the cost $C(S - T_i)$ of the set $S - T_i$ of objects not responding to the test. Treatments terminate action on the objects of T_i, $m < i \le n$, (i.e., treat them) so the only objects needing further action are the objects in $S - T_i$; we add in the cost $C(S - T_i)$. The essence of an argument by induction that C(S) is correctly computed uses the assumption that $C(S \cap T_i)$ and $C(S - T_i)$ are the correct costs for the subtrees and then we note from the above description that the correct minimum is taken to compute C(S).

2. The Boolean Vector Machine

The BVM is a CCC parallel machine. It is a typical ultracomputer[14]. Since the BVM communication network resembles the Benes permutation network, it can accomplish any permu-

tation within O(log n) time if the control bits are precalculated [10],[13]. Logically the BVM can be viewed as a bit array shown in Fig. 2.

	PE 0	PE 1	PE 2	PE 3	PE 4	PE 5	PE 6	PE 7	PE 8	...	...
Reg. A	x	x	x	x	x	x	x	x	x	...	...
Reg. B	x	x	x	x	x	x	x	x	x	...	...
Reg. R[0]	x	x	x	x	x	x	x	x	x	...	...
Reg. R[1]	x	x	x	x	x	x	x	x	x	...	...
Reg. R[2]	x	x	x	x	x	x	x	x	x	...	...
Reg. R[3]	x	x	x	x	x	x	x	x	x	...	...
Reg. R[4]	x	x	x	x	x	x	x	x	x	...	...
Reg. R[5]	x	x	x	x	x	x	x	x	x	...	...
Reg. R[6]	x	x	x	x	x	x	x	x	x	...	...
...		.	.	.		.	.	.		...	...
...		.	.	.		.	.	.		...	...

Fig. 2.

Each row of the bits forms a register. Each column forms a PE. Let r be a positive integer and $Q=2^r$, there are total 2^{r+Q} PE's, as required by a complete CCC network. The number of registers L depends on the BVM implemented. Our BVM has L=256 registers.

Each of the group of PE's $(2^r*i,...,2^r*i+j, ...,2^r*i+2^r-1)$, $0\le i<2^Q$, form a cycle, thus the address of PE 2^r*i+j can be represented alternatively as (i, j) with the first component being the cycle number and the second the address within the cycle. Within cycle i PE (i, j) is only connected to its predecessor $(i,(j+Q-1)\%Q)^*$ and its successor (i, (j+1)%Q). In addition each PE (i, j) is connected to its lateral neighbor $(i\hat{}2^j, j)$, the cycles are thus connected together.

The BVM is a bit-oriented machine. Only Boolean function operations are allowed. Each of its instruction involves possibly register A and B and at most another register. Its instruction has the form:

{A or R[j]}, B = f, g(F, D, B) {IF or NF} <set>;

Two assignment operations will be simultaneously performed by executing this instruction. The first assigns f(F, D, B) to either A or R[j], the second assigns g(F, D, B) to B. f and g are any

* As in the C language[5], %, /, &, |, ^ are the modulo, integer division, and, or, and exclusive-or operations respectively

Boolean functions of three arguments. F may be A or R[j]. D may be A.N or R[j].N. N denotes a neighbor PE of PE (c, p). It can be:

S: successor PE $(c, (p+1)\%Q)$;

P: predecessor PE $(c, (p+Q-1)\%Q)$;

L: lateral PE $(c \hat{} 2^r, p)$,

XS: even successor exchange PE $(c, p \hat{} 2^0)$;

XP: even predecessor exchange XP=P if p is even;

XP=S if p is odd;

I: input one bit to PE (0, 0), PE $(2^Q-1, Q-1)$ outputs one bit at the same time. All other PEs get bits from their predecessors except PEs (., 0), which get bits from PEs (.-1, Q-1).

The {IF or NF} <set> denotes the activate/deactivate set. <set> is a subset of {0, 1, ..., 2^r-1}. IF <set> means all the PE's (i, j), $0 \le i < 2^Q$ and $j \in <set>$, will be activated while the remaining PE's will be deactivated. The meaning of NF <set> is just the opposite. If the part {IF or NF} <set> is not present in the instruction, then all the PE's are activated.

There is a special register, register E. which is used as an enable/disable register. PE i will be enabled or disabled according to whether its bit of E register is 1 or 0. E register itself is always enabled.

The value of PE's will not be affected(except that of register E) if it is deactivated or disabled.

For further details of the BVM, the reader is referred to [15], [16].

8. Hypercube Algorithm

A hypercube connection network has been suggested [12] as a network for connecting together an array of PEs. The hypercube connection network connects PE x to any PE whose address differs from x in exactly one bit. Thus a machine of 2^r PEs will have each PE connected to r PEs. Since the hypercube network seems to be more regular than the CCC, and each PE has

more connections to other PEs, in many situations designing a hypercube algorithm is more convenient and straightforward than designing a CCC network algorithm. Unfortunately with n PEs a hypercube network requires about $n \log n / 2$ links. With a CCC connection only about $3n/2$ links are needed.

An algorithm is in the ASCEND(DESCEND)[13] form if it consists of a sequence of basic operations on pairs of data, where the addresses of the pairs differ successively in bit 0, bit 1, ..., bit p-1 (bit p-1, bit p-2, ..., bit 0), here and henceforth bits are counted from the least significant bit.

Preparata and Vuillemin showed in [13] that the ASCEND/DESCEND algorithms can be executed on the CCC network fairly efficiently. Precisely speaking, these hypercube network algorithms can be simulated on a CCC at a slowdown of a factor of 4 to 6, regardless of the network sizes. Thus designing an ASCEND/DESCEND algorithm for a hypercube, and transforming it into a CCC algorithm seems to be a reasonable way of designing an efficient CCC algorithm.

In the CCC the links connecting the lateral PEs are called highsheaves. Highsheaves correspond to the high-order bit connections in the hypercube. The number of high-order bits is Q, the number of bits of the cycle number. The lowsheaves are virtual links in the CCC which correspond to the low-order bit connections in the hypercube. There are r bits for the lowsheaves. The lowsheaves connections in the CCC is formed by shuffling or unshuffling data inside cycles.

4. Several Important BVM Algorithms

Several important BVM algorithms are presented here. These algorithms are useful in programming the BVM. The cycle-ID and the processor-ID are the most basic modules which are used in almost all BVM algorithms. Broadcasting and propagation algorithms handle the typical dataflow patterns on the BVM. These algorithms or their adapted version are used to construct the test-and-treatment algorithm. As we present these algorithms, we'll also discuss how to generate the control bits for these algorithms. Although these control bits can be precalculated, it

will save the precalculation time and the runtime storage when these control bits are generated on the fly.

1. Cycle-ID

There are two ways of viewing the cycle-ID. The first is that PE (i, j) holds the j-th bit of cycle number i. Thus the bits held by all the PEs in cycle i form the cycle number i. In the CCC network the lateral links correspond to the high-order bit links in the hypercube. The alternative way of viewing the cycle-ID is that the bit of the cycle-ID PE i holds is 1 iff PE i is at the 1-end of its lateral link. For the CCC with n=64 PEs, the cycle-ID is shown in Fig. 3.

PE 0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
PE 1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
PE 2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
PE 3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c
	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y
	c		c	c	c	c	c	c	c	c	c	c	c	c	c	c
	l	l	l	l	l	l	l	l	l	l	l	l	l	l	l	l
	e	e	e	e	e	e	e	e	e	e	e	e	e	e	e	e
	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5

Fig. 3.

the digit at cycle i and PE j represents the bit held by PE j in cycle i.

In [15] the following algorithm is given for generating the cycle-ID. The time complexity is O(logn).

```
cycle-ID()
{
    A=1;
    A=A.I;  /* input a bit 0 to PE 0 */
    for(i=1; i<Q; i++) {
       A=A & A.L;
       A=A.I;
    }
    A=A.P;
    for(i=1; i<Q; i++) {
       A=A & A.L;
       A=A.P;
    }
}
```

2. Processor-ID

The processor-ID is defined as the pattern of addresses such that each PE holds its own address. For 8 PEs the processor-ID is shown in Fig. 4.

	PE 0	PE 1	PE 2	PE 3	PE 4	PE 5	PE 6	PE 7
R[i]	0	0	0	0	1	1	1	1
R[i+1]	0	0	1	1	0	0	1	1
R[i+2]	0	1	0	1	0	1	0	1

Fig. 4.

The algorithm for generating the processor-ID is as follows:

```
Processor-ID()
{
 1. R[S]=cycle-ID();
 2. for(i=1; i<Q; i++) {
      R[S+i]=R[S+i-1];
      R[S+i]=R[S+i].S;
    }
 3. for(i=0; i<Q/2; i++)
      for(j=0; j<Q; j++) {
        if(i%2==0) R[S+j]=R[S+j].XP IF {e | i<e<Q-i};
        else R[S+j]=R[S+j].XS IF {e | i<e<Q-i};
      }
 4. for(i=0, i<Q; i++)
      for(j=0; j<r; j++)
        R[S-j-1]=bit(r-j-1, i) IF { i };
}
```

Recall that R[i].S, R[i].XS, R[i].XP are the R[i]'s successor, even successor exchange and even predecessor exchange respectively. The function bit(p, q) is the p-th bit of q. The timing is $O(\log^2 n)$. Here we give an example of the execution of the algorithm. In Fig. 5, (i) is the pattern created after the execution of the statement labeled i in the algorithm.

3. Broadcasting

The broadcasting algorithm is used to broadcast data from one PE to all the other PEs. In [9] Nassimi and Sahni studied data broadcasting on SIMD machines, here we only consider the broadcasting on the BVM. This algorithm and the algorithms for propagation will be presented as ASCEND hypercube algorithms, for the transformed BVM algorithms look much complicated and

S	0000	1000	0100	1100	0010	1010	0110	1110 ...

(1)

S	0000	1000	0100	1100	0010	1010	0110	1110 ...
S+1	0000	0001	1000	1001	0100	0101	1100	1101 ...
S+2	0000	0010	0001	0011	1000	1010	1001	1011 ...
S+3	0000	0100	0010	0110	0001	0101	0011	0111 ...

(2)

S	0000	1111	0000	1111	0000	1111	0000	1111 ...
S+1	0000	0000	1111	1111	0000	0000	1111	1111 ...
S+2	0000	0000	0000	0000	1111	1111	1111	1111 ...
S+3	0000	0000	0000	0000	0000	0000	0000	0000 ...

(3)

S-2	0101	0101	0101	0101	0101	0101	0101	0101 ...
S-1	0011	0011	0011	0011	0011	0011	0011	0011 ...
S	0000	1111	0000	1111	0000	1111	0000	1111 ....
S+1	0000	0000	1111	1111	0000	0000	1111	1111 ...
S+2	0000	0000	0000	0000	1111	1111	1111	1111 ...
S+3	0000	0000	0000	0000	0000	0000	0000	0000 ...

(4)

Fig 5. An instance of n=64.

provide little insight into the algorithms. Let the total number of PEs be 2^m. Let j<i> denote the i-th bit of j and j#i denote the binary number which is obtained by complementing the i-th bit of j. The following algorithm broadcasts the content of PE 0 to all the PEs.

```
Broadcasting()
{
 SENDER=0;
 SENDER(PE[0])=1;
 for(i=0; i<m; i++)
   forall PE[j];
    if 1-END(PE[j], i) && SENDER(PE[j#i])
     {
      PE[j] = PE[j#i];
      SENDER(PE[j]) = SENDER(PE[j#i]);
     }
}
```

SENDER is a register. SENDER(PE[0]) is the bit belonging to both register SENDER and PE 0. 1-END(PE[j], i) is true when i-th bit of j is 1. For a 16-PE array, the broadcasting process is shown in Fig. 6.

1. 0000 -> 0001,
2. 0000 -> 0010, 0001 -> 0011,
3. 0000 -> 0100, 0001 -> 0101,
   0010 -> 0110, 0011 -> 0111,
4. 0000 -> 1000, 0001 -> 1001,
   0010 -> 1010, 0011 -> 1011,
   0100 -> 1100, 0101 -> 1101,
   0110 -> 1110, 0111 -> 1111.

Fig. 6.

The algorithm is an ASCEND algorithm, thus O(m) time is enough to execute the transformed algorithm on a 2^m-PE CCC computer.

To run this algorithm on the BVM, the control bits must be considered. The approach we choose works as follows. First an arbitrary register SENDER in the BVM is chosen. Set every bit of SENDER to 0 by using one instruction. Then input a bit 1 to the bit belonging to both PE[0] and register SENDER. Afterwards this bit will be broadcast in the instruction PE[j]=PE[j#i], and the content of register SENDER will be used to identify the sender. If the number of bits to be broadcast is k, then the algorithm takes O(km) time.

4. Propagation

Propagation refers to propagate data from one set of PEs to another set. We consider two kinds of propagation here.

The i-PE group is the set of PEs whose addresses have exactly i 1's. The first kind of propagation considered here propagates data from the N-PE group to the (N+1)-PE group such that PE j in the (N+1)-PE group receives data from PE k in the N-PE group iff when k<i>=1 then j<i>=1.

Example:

N=2. For the 16-PE array, PE 0111 receives data from

PE 0110, 0101 and 0011. #

The algorithm is shown below:

```
Propagation1()
{
    for(i=0; i<m; i++)
      forall PE[j]
        if SENDER(PE[j#i]) && 1-END(PE[j], i)
          PE[j] = COMBINE(PE[j], PE[j#i]);
}
```

The time complexity is O(m). If the propagation needed requires data to flow through the 0-PE group, the 1-PE group, ..., the m-PE group, then the algorithm must be used m times and the timing will be $O(m^2)$.

It might seem possible to enable all the PEs in the (N+1)-PE group, thus allowing the propagation to be done more naturally. However in many situations in which the algorithm is used, data are propagated from the 0-PE group to the 1-PE group to the 2-PE group,..., to the N-PE group, but initially no PE knows which group it belongs to except the PE in the 0-PE group which is PE 0. Thus a PE in the (N+1)-PE group will know that it is in the (N+1)-PE group from the fact that the sender is in the N-PE group and it itself is at the 1-end of the communication link. Certainly one can generate the processor-ID and count the number of 1's in it to decide to which group each PE belongs, but that involves more overhead.

Another kind of propagation requires data to propagate from the N-PE group to the M-PE group, where without loss of generality we assume N < M. PE k in the M-PE group will get the data from PE j in the N-PE group such that if j<i>=1 then k<i>=1.

Example:

M=3, N=1, for the 16-PE array, PE 0111 will get data from PE 0001, 0010, 0100. #

Though the first propagation algorithm may be used for this purpose, it is relatively slow. The alternative way to do it is shown in the following example.

Example:

n=4, data will propagate from the 1-PE group to the 4-PE group.

Initially data are in PEs 0001, 0010, 0100, 1000.

1. 0001 does not transmit. 0010 -> 0011, 0100 -> 0101, 1000 -> 1001.

2. 0001 -> 0011, 0011 does not transmit, 0101 -> 0111, 1001 -> 1011, 0011 combines the data coming from 0001 and the data inside itself.

3. 0011 -> 0111, 0111 does not transmit, 1011 -> 1111, 0111 combines the data coming from 0011 and the data inside itself.

4. 0111 -> 1111, 1111 does not transmit, it combines the data coming from 0111 and the data inside itself. #

Now we shall discuss the generation of the control bits. We use one bit in each PE to indicate if it is a legal sender. Only PEs with this bit set have the right to send. In the sending process the sender will not discard this bit. The receiver acquiring this bit will become a legal sender. If the receiver gets two bits from two senders, it must combine the data and the control bits (using a logical or operation). Notice that this is different from the scheme in the propagation of the first kind, since here immediately after the receiver gets the data it becomes a sender. In the propagation of the first kind, a sender or a receiver remains the same until all the PEs in the (i + 1)-PE group have been connected to the PEs in the i-PE group.

Initially PEs in the N-PE group are identified as senders. Only PEs in the higher PE groups can get data from lower PE groups. To control the direction of the dataflow on the BVM the cycle-ID should be used.

The algorithm is as follows :

```
Propagation2()
{
   for(i=0; i<m; i++)
     forall PE[j];
       if SENDER(PE[j#i]) && 1-END(PE[j], i)
         {
           PE[j] = COMBINE(PE[j], PE[j#i]);
           SENDER(PE[j]) = SENDER(PE[j#i]);
         }
}
```

The timing is O(m)=O(logn).

5. A Parallel Algorithm for the Test-and-Treatment Problem

In actual computation we'll assign an array M[S, k] to calculate C(S): $M[S,i]=t_i p(S)+C(S\bigcap T_i)+C(S-T_i), 0\le i\le m$, and $M[S,i]=t_i p(S)+C(S-T_i), m<i<N$, therefore

$$C(S)=min\{M[S,i] \mid 0\le i<N\}.$$

The parallel algorithm is given below:

```
TT(S, T, P, t)
{
   foreach i: 0<=i<N do {
      TP[S,i]=t_i * p(S), if #S > 0,

      M[S,i] = { 0, if #S=0.
               { INF, otherwise.
   }

   for(j=1; j<=#U; j++) {
      foreach (S, i): U ⊇ S and #S=j and 0<=i<N do {
        M[S,i]=M[S-T_i,i];
        M[S,i]+=TP[S,i];
        if(i <= m) M[S,i]+=M[S⋂T_i,i];
      }
      foreach (S,i): U ⊇ S and #S=j and 0<=i<N do
        M[S,i]=min(M[S,x] | 0<=x<N);
      }
   }
 }
```

#S in the algorithm denotes the size of set S.

Note that the conditions $S\bigcap T_i\neq\phi$ and $S-T_i\neq\phi$ will be taken into account automatically. If $S\bigcap T_i=\phi$ and T_i is a test, then $S-T_i=S$. Since S is initialized to infinity(INF), we have:

$$M[S,i]=t_i p(S)+C(S\bigcap T_i)+C(S-T_i)=t_i p(S)+C(S)=INF.$$

So it will be excluded in the minimization or C(S) will be INF depending on whether or not there is a M[S,j] such that M[S,j]<INF. The same reasoning applies if T_i is a treatment or if $S-T_i=\phi$(then $S\bigcap T_i=S$).

The j-loop is needed because when we calculate M[S,i] we should have all the $C(S\bigcap T_i)$

and $C(S-T_i)$ available.

6. The ASCEND/DESCEND Algorithm

Can our parallel TT algorithm be transformed into the ASCEND/DESCEND form? Observe that if we assign a PE to each (S, i) pair then M[S,i] and TP[S,i] are placed at the different sections of the same PE, thus the instruction M[S,i] +=TP[S,i] can be executed in parallel by all PEs at once. However the instruction $M[S,i]=M[S-T_i,i]$, $M[S,i]+=M[S\bigcap T_i,i]$ and the minimization require communications between different PEs.

The minimization part of the algorithm can be transformed into the following ASCEND form:

```
for(t=0; t<log N; t++)
   foreach(S,i): U ⊇ S and #S=j and 0≤i<N do
     M[S,i]=min(M[S,i], M[S, i#t]);
```

where i#t is the binary number obtained by complementing the t-th bit (from right) of i.

Suppose $N=2^p$ (otherwise we let $T_N=T_{N+1}=...=T_{2^p-1}=U$ and all of them will be treatments with cost INF), after executing M[S,i]=min(M[S,i], M[S,i#0]),

M[S,j]=M[S,j+1]=min(M[S,j], M[S,j+1]), where j=0, 2,..., N-2.

Assume after executing M[S,i]=min(M[S,i], M[S,i#(q-1)]):

M[S,j]=M[S,j+1]=...=M[S,$j+2^q-1$]=

min(M[S,j], M[S,j+1],..., M[S,$j+2^q-1$]),

for j: $j/2^{q-1}$ is even.

After executing M[S,i]=min(M[S,i], M[S, i#q]) M[S,j]=M[S,j+2^q]=min(M[S,j], M[S,j+2^q]), for j: j/(2^q) is even and $0\le j<N$. By induction we know now: M[S,j]=M[S,j+1]=...=M[S,j+2^{q+1}-1]=min(M[S,j], M[S,j+1],..., M[S,j+2^{q+1}-1].

Let q=p-1, all the PEs associated with set S will get the minimum value. Fig. 7 shows an example with p=3.

Now consider the instruction:

i	initial value	t=	1	2	3
0	5		1	1	0
1	1		1	1	0
2	6		2	1	0
3	2		2	1	0
4	3		3	0	0
5	4		3	0	0
6	0		0	0	0
7	7		0	0	0

Fig. 7.

foreach (S,i) M[S,i]=M[$S-T_i$,i].

Can this operation be transformed into the ASCEND/DESCEND form?

Let us begin by expanding it into its component operations as follows:

```
foreach (S,i) do {
   R[S,i]=M[S-Ti,i];
   M[S,i]=R[S,i];
 }
```

This emphasizes that, if a unique PE holds each element of M[S,j], some communication between PEs is needed. The variable R is introduced to handle the possibility that the PEs involved are not in fact neighbors, so that each item of information must be passed along a chain of PEs before it reaches its destination. For simplicity, we also assume that R[S,i] is located somewhere in the memory of the same PE that also holds M[S,i], for each distinct pair (S,i).

Consider now the operation

$R[S,i]=M[S-T_i,i]$.

For each S and i, this operation is well-defined, since the set $S-T_i$ is uniquely determined by S and T_i, thus ensuring that each PE which receives information during this activity receives information from only one PE. However, the converse is not true: a given PE may send its information to several others, as the example shown in Fig. 8.

Thus $M[\phi,i]$ will send its value to $R[\phi,i]$, R[{0},i], R[{1},i] and R[{0,1},i], and M[{2},i] will send its value to the other four PEs. In general, $M[S-T_i,i]$ must be broadcast to $R[(S-T_i)\bigcup V,i]$, for each V such that $S\bigcap T_i \supseteq V$. The following loop accomplishes the

U={0,1,2}, T={0,1}.

If S is:	S-T is:
ϕ	ϕ
{0}	ϕ
{1}	ϕ
{1,0}	ϕ
{2}	{2}
{2,0}	{2}
{2,1}	{2}
{2,1,0}	{2}

Fig. 8.

required broadcast, for all i, $0 \leq i < N$:

```
R[S,i]=M[S,i];
for(e=0; e<k; e++)
  foreach (S,i): U ⊇ S and 0≤i<N and e ∈ S ⋂ Ti do
    R[S,i]=R[S-{e},i];
M[S,i]=R[S,i];
```

Continuing the previous example, Fig. 9 shows the value of Q_e just after the e-th iteration of the loop.

S	e: -	0	1	2
ϕ	ϕ	ϕ	ϕ	ϕ
{0}	{0}	ϕ	ϕ	ϕ
{1}	{1}	{1}	ϕ	ϕ
{1,0}	{1,0}	{1}	ϕ	ϕ
{2}	{2}	{2}	{2}	{2}
{2,0}	{2,0}	{2}	{2}	{2}
{2,1}	{2,1}	{2,1}	{2}	{2}
{2,1,0}	{2,1,0}	{2,1}	{2}	{2}

Fig. 9.

Let $I_t = \{j \in U \mid j \leq t\}$, then just before e takes on value t, $R[(S-T_i) \bigcup (S \bigcap T_i \bigcap I_{t-1}), i]$ holds $M[S-T_i, i]$. For this is true initially, before e takes on value 0, since $I_{-1}=\phi$, and $R[S-T_i, i]$ holds $M[S-T_i, i]$. Assuming the statement was true before e was set to t, the loop body, executed with e set to t causes certain elements of array R to change in value. Specifically, whenever $t \in S \bigcap T_i$, R[S,i] is replaced by R[S-{t},i]. Suppose $t \notin S \bigcap T_i$ then $S \bigcap T_i \bigcap I_t = S \bigcap T_i \bigcap I_{t-1}$, and $R[(S-T_i) \bigcup (S \bigcap T_i \bigcap I_t), i]$ is unchanged, and held $M[S-T_i, i]$ prior to this execution of the loop body, by the inductive assumption. Suppose instead that

$t \epsilon S \bigcap T_i$. Then $R[(S-T_i)\bigcup(S \bigcap T_i \bigcap I_t), i]$ is changed to this iteration of the loop.

Since $X \bigcap I_t = (X \bigcap I_{t-1}) \bigcup (X \bigcap \{t\})$, and $t \epsilon S \bigcap T_i$, $S \bigcap T_i \bigcap I_t = S \bigcap T_i \bigcap I_{t-1} \bigcup \{t\}$. Because $t \epsilon S \bigcap T_i$, $t \epsilon T_i$. thus $t \overline{\epsilon} S - T_i$, and $(S-T_i)\bigcup(S \bigcap T_i \bigcap I_t) - \{t\} = (S-T_i)\bigcup(S \bigcap T_i \bigcap T_{t-1})$, by the inductive assumption, $R[(S-T_i)\bigcap(S \bigcap T_i \bigcap I_{t-1}), i] = M[S-T_i, i]$. Hence, after the assignment, $R[(S-T_i)\bigcap(S \bigcap T_i \bigcap I_t), i] = M[S-T_i, i]$ also.

Finally, after all iterations of the loop on e, $I_t = U$, and $S \bigcap T_i \bigcap I_t = S \bigcap T_i$. Thus, for all S and i, $R[S, i] = M[S-T_i, i]$.

Similarly, the operation

$$if\ (i \leq m)\ M[S, i] += M[S \bigcap T_i, i]$$

can be resolved into:

```
Q[S,i]=M[S∩T_i,i];
if(i<=m) M[S,i]+=Q[S,i];
```

Again, concentrating on the operation R[S,i]=$M[S \bigcap T_i, i]$, we observe that value $M[S \bigcap T_i, i]$ can be broadcast to each PE holding $Q[V \bigcup (S \bigcap T_i), i]$, for every V such that $S - T_i \supseteq V$. By symmetry with the previous argument, the following loop performs this operation:

```
Q[S,i]=M[S,i];
for(e=0; e<k; e++)
   foreach (S,i): U ⊇ S and 0≤i<N and e ∈ S-T_i do
      Q[S,i]=Q[S-{e},i];
```

The complete algorithm now appears as:

```
TT()
{
    foreach i: 0<=i<N do {
       if(#S>0) TP[S,i]=t_i *p(S);
       M[φ,i]=0;
       if(#S>0) M[S,i]=INF;
    }

    for(j=1; j<=k; j++) {
       foreach (S,i): P_1(S,i) {
         Q[S,i]=R[S,i]=M[S,i];
       }
       for(e=0; e<k; e++) {
         foreach (S,i): P_1(S,i) and e ∈ S ∩ T_i {
           R[S,i]=R[S-{e},i];
         }
         foreach (S,i): P_1(S,i) and e ∈ S - T_i {
           Q[S,i]=Q[S-{e},i];
         }
       }

       foreach (S,i): P(S,i,j) {
         M[S,i]=R[S,i];
         M[S,i]+=TP[S,i];
         if(i<=m) M[S,i]+=Q[S,i];
       }
       for(t=0; t<log N; t++)
         foreach (S,i): P(S,i,j) {
           M[S,i]=min(M[S,i],M[S,i#t]);
         }
    }
}
```

Where $P_1(S,i) \equiv U \supseteq S$ and $0 \le i < N$, and $P(S,i,j) \equiv P_1(S,i)$ and #S=j.

7. Implementation Schemes

On the BVM each PE will stand for a pair (i, j), where i and j are binary numbers and ij, the concatenation of i and j, is the address of the PE. |i|, the number of bits in i, is k. The component i denotes a subset S of U, a ∈ S iff a-th bit of i is 1. j is the index of a test or a treatment.

Example:

> k=4, PE 011011 stands for (0110, 11), the set S denoted by i is {1, 2}, the index number of the test or the treatment is 3 #

The activate/deactivate mechanism is very convenient, unfortunately it can only provide limited masking capabilities. The enable register can provide any kind of enable/disable patterns, but generating these control bits is difficult. Here we show how these control bits are generated and how the algorithm is implemented by using the algorithms introduced in section 5.

The predicates $e \epsilon S \bigcap T_i$ and $e \epsilon S - T_i$ can be implemented by using the processor-ID. The processor-ID bits will let each PE know the set S it represents. T_i should be input to the BVM. The most interesting part of the algorithm is the loop indexed by the variable e. The technique used here is the one we introduced in the propagation algorithm. Note that by imposing the conditions $e \epsilon S \bigcap T_i$ and $e \epsilon S - T_i$ the result becomes $R[S,i]=R[S-T_i,i]$ and $Q[S,i]=Q[S \bigcap T_i,i]$. The dataflow is controlled by the predicate $e \epsilon S \bigcap T_i$ and $e \epsilon S - T_i$. Because each iteration of the loop indexed by j will increment the size of the sets #S by 1, the predicate P(S, i, j) can be implemented by using the propagation of the first kind. The cycle-ID will be used in the propagation algorithm.

8. Conclusion

Many NP-complete problems can be solved on the BVM fairly efficiently, as we illustrate using the test-and-treatment problem. Indeed, the test-and-treatment problem itself is of real interest as it has many important applications. A parallel algorithm for this problem is presented and implemented on the Boolean Vector Machine. The communication problem and the PE allocation problem have been solved so that a speedup $O(\frac{p}{\log p})$ is achieved. Algorithms used in constructing the test-and-treatment algorithm have also been presented and reveal in some degree the different methods of programming serial and parallel machines.

References

[1] Garey, M.R. Optimal binary identification procedures. SIAM J. Appl. Math. Vol. 23, No. 2, Sept. 1972. pp. 173-186.

[2] Garey, M.R. and Graham, R.L. Performance bounds on the splitting algorithm for binary

testing. Acta Informatica 3, 347-355(1974).

[3] Hyafil, L. and Rivest, R.L. Constructing optimal binary decision trees is NP-complete. Inf. Process. Lett. 5, 15-17(1976).

[4] Karnin, E.D. A parallel algorithm for the knapsack problem. IEEE Tran. comput. Vol. C-33, No. 5, May 1984. pp. 404-408.

[5] Kernighan, B.W. and Ritchie, D.M. The C Programming Language. Prentice-Hall (1978).

[6] Loveland, D.W. Selecting Optimal Test Procedures from Incomplete Test Sets. Proc. First Int. Symp. Policy Anal. Inf. Sci., pp. 228-235. Duke University, Durham, NC, 1979.

[7] Loveland, D.W. Performance Bounds for Binary Testing with Arbitrary Weights. Acta Informatica 22, 101-114(1985).

[8] Mead, C. and Conway L. Introduction to VLSI systems. Addison-Wesley(1980). pp. 305-313.

[9] Nassimi, D. and Sahni, S. Data Broadcasting in SIMD Computers. IEEE Trans. Computer, Vol.C-30, No. 2, Feb. 1981.

[10] Nassimi, D. and Sahni, S. Parallel Algorithms to Set Up the Benes Permutation Network. IEEE Trans. Computer, Vol. C-31, No. 2, Feb. 1982.

[11] Payne, R.W. and Preece, D.A. Identification keys and diagnostic tables: a review. Jour. of the Royal Stat. Soc. (Series A) 143(3), 253-242 (1980).

[12] Pease, M.C. The indirect binary n-cube microprocessor array. IEEE Trans. Comput. C-26, 5 (May 1977), 458-473.

[13] Preparata, F.P. and Vuillemin, J. The Cube-Connected Cycles: A Versatile Network for Parallel Computation. CACM 24,5 (May 1981), 300-309.

[14] Schwartz J.T. Ultracomputers. ACM Trans. Programming Languages and Systems, Vol. 2, No 4, (Oct. 1980), 484-521.

[15] Wagner, R.A. A Programmer's View of the Boolean Vector Machine, Model-2. CS-1981-8, Dept of Computer Science, Duke Univ., Oct. 1981.

[16] Wagner, R.A. The Boolean Vector Machine (BVM). IEEE 1983 Conf. Proc. of 10-th Ann. International Symposium on Computer Architecture, pp. 59-66.

END

FILMED

1-86

DTIC